

```
////////////////////////////////////
//
// Title       : ColorMatch.c
// Author      : Siming Lin
// Date       : 
// Copyright   : National Instruments . All Rights Reserved.
// Access      : Company Confidential
// Purpose     : Implements the ColorMatch function.
//
////////////////////////////////////
```

```
//=====
// Include Directives
//=====
#include PRECOMP
#include "Analysis/Color/CommonColor.h"
#include <stdlib.h>
#include <math.h>
```

```
////////////////////////////////////
//
// ColorMatch
//
// Description:
//   Match the color feature between the input color image and input color feature
//   array
//   Feature:
//   the estimated color spectrum
// Parameters:
//   pSrcRef - source image
//   pRoi - ROI
//   pColSpectrumRef - input spectrum feature
//   pTolerance - match threshold
//   pSatThreshold - threshold for distinguishing two color with the same hue value
//   MatchFlag - Matching indicator
//   MatchScore - Match score
// Return Value:
//   Error code
//
////////////////////////////////////
```

```
// algorithms for color feature extraction
```

```
enum ColorFeatureSet{
    kColSpectrum = 0,
    kColSignature,
    kColHisIntersection
};
```

```
enum ColorComplexity{
    kComplexityLow = 0,
    kComplexityMedium,
    kComplexityHigh
};
```

```
enum ColorBinNumber {
    kColorBinNbLow = 16,
    kColorBinNbMedium=30,
    kColorBinNbHigh=58
};
```

```

#define ColorSatSpread 0.2
#define ColorSatRemain 0.8
#define BWHueThreshold 10
#define BlkLgtThreshold 5
#define BWLgtMiddle 128
#define OffsetColor 18 // the center of the hue value of red color

GRLIBError ColorMatch(const GRImage* pSrcIMRef, Array1D* pColSpectrumRef, const ROI* pRoi,
, double* ColTolerance, int pSatThreshold, Array1D* pMatchFlagRef, Array1D*
pMatchScoreRef) {

    GRImage* lMaskImagePtr = NULL;
    MaskPart lMaskPart;
    const Pix8* lMaskPixPtr;
    const Pix8* lMaskLinePtr;
    int lpixelsOutsideMask=0;

    const GRImage* lSrcIMPtr;
    ConstPixelPtr lSrcPixPtr;

    const PixRGB *lSrcPixRGBPtr, *lBufSrcRGBPtr;

    const PixHSL *lSrcPixHSLPtr, *lBufSrcHSLPtr;

    double *lColImSpecA1DWPtr, *lColTempSpecA1DWPtr;
    long *lMatchFlagPtr, *lMatchScorePtr;

// Coord lCoord;

    int i, jj, lSizeXLW, lSizeY LW, lSrcLineWidth, lCollW, lLineLW ;

    double lImageAreaDW, lHueStepDW;

    int lColSpectrumDimLW, lNbColFeature, lNbHueBin, lHueIndex,
lNumColorEntry=0;

    long lRHHVal, lGSSVal, lBLVVal;
    long lhue, lsat, llgt, lBlkThreshold;
    Byte lreplace = 191, offset=0;

    long lcoring = 0;
    double h,s,l;
    long lMskLineWidth, lNbROIContour, lROIContourIndex;

    double lColDiff, lColImSpreaded[60], lColTempSpreaded[60], lColImSpreaded2[60],
lColTempSpreaded2[60];

    GRLIBError error = ERR_SUCCESS;

    // lookup table for function f(x)=exp(-0.025*x), x=0 ~ 200
    static double expLookup[] = {
        1.0000, 0.9753, 0.9512, 0.9277, 0.9048, 0.8825, 0.8607, 0.8395, 0.8187, 0.
7985,
        0.7788, 0.7596, 0.7408, 0.7225, 0.7047, 0.6873, 0.6703, 0.6538, 0.6376, 0.
6219,
        0.6065, 0.5916, 0.5769, 0.5627, 0.5488, 0.5353, 0.5220, 0.5092, 0.4966, 0.
4843,
        0.4724, 0.4607, 0.4493, 0.4382, 0.4274, 0.4169, 0.4066, 0.3965, 0.3867, 0.
3772,

```

```
2938, 0.3679, 0.3588, 0.3499, 0.3413, 0.3329, 0.3247, 0.3166, 0.3088, 0.3012, 0. ✓
2288, 0.2865, 0.2794, 0.2725, 0.2658, 0.2592, 0.2528, 0.2466, 0.2405, 0.2346, 0. ✓
1782, 0.2231, 0.2176, 0.2122, 0.2070, 0.2019, 0.1969, 0.1920, 0.1873, 0.1827, 0. ✓
1388, 0.1738, 0.1695, 0.1653, 0.1612, 0.1572, 0.1534, 0.1496, 0.1459, 0.1423, 0. ✓
1081, 0.1353, 0.1320, 0.1287, 0.1256, 0.1225, 0.1194, 0.1165, 0.1136, 0.1108, 0. ✓
0842, 0.1054, 0.1028, 0.1003, 0.0978, 0.0954, 0.0930, 0.0907, 0.0885, 0.0863, 0. ✓
0655, 0.0821, 0.0801, 0.0781, 0.0762, 0.0743, 0.0724, 0.0707, 0.0689, 0.0672, 0. ✓
0510, 0.0639, 0.0623, 0.0608, 0.0593, 0.0578, 0.0564, 0.0550, 0.0537, 0.0523, 0. ✓
0398, 0.0498, 0.0486, 0.0474, 0.0462, 0.0450, 0.0439, 0.0429, 0.0418, 0.0408, 0. ✓
0310, 0.0388, 0.0378, 0.0369, 0.0360, 0.0351, 0.0342, 0.0334, 0.0325, 0.0317, 0. ✓
0241, 0.0302, 0.0295, 0.0287, 0.0280, 0.0273, 0.0266, 0.0260, 0.0253, 0.0247, 0. ✓
0188, 0.0235, 0.0229, 0.0224, 0.0218, 0.0213, 0.0208, 0.0202, 0.0197, 0.0193, 0. ✓
0146, 0.0183, 0.0179, 0.0174, 0.0170, 0.0166, 0.0162, 0.0158, 0.0154, 0.0150, 0. ✓
0114, 0.0143, 0.0139, 0.0136, 0.0132, 0.0129, 0.0126, 0.0123, 0.0120, 0.0117, 0. ✓
0089, 0.0111, 0.0108, 0.0106, 0.0103, 0.0101, 0.0098, 0.0096, 0.0093, 0.0091, 0. ✓
0069, 0.0087, 0.0084, 0.0082, 0.0080, 0.0078, 0.0076, 0.0074, 0.0073, 0.0071, 0. ✓
0054, 0.0067, 0.0066, 0.0064, 0.0063, 0.0061, 0.0059, 0.0058, 0.0057, 0.0055, 0. ✓
```

```
};
```

```
// get the pointer to the input image structure
lSrcIMPtr = pSrcIMRef;
lSrcLineWidth = lSrcIMPtr->lineWidth;
if( pRoi->numContours >0 )
    lNbROIContour=(long) pRoi->numContours; // ROI connected
else
    lNbROIContour=1; // ROI not connected, use the entire image

// Set the size of match flag array
if (error = ResizeArray1D(pMatchFlagRef, sizeof(long), lNbROIContour))
{
    goto END;
}

// Get the match flag array pointer
if (error = GetArray1DPtr(pMatchFlagRef, &lMatchFlagPtr))
{
    goto END;
}
```

```
// Set the size of match Score array

if (error = ResizeArray1D(pMatchScoreRef, sizeof(long), lNbROIContour))
{
    goto END;
}

// Get the match score array pointer

if (error = GetArray1DPtr(pMatchScoreRef, &lMatchScorePtr))
{
    goto END;
}

/*Initilization */

for (i=0; i<lNbROIContour; i++) {

    lMatchScorePtr[i]=0;
    lMatchFlagPtr[i]=0;

}

// Get the color array pointer

if (error = GetArray1DPtr(pColSpectrumRef, &lColTempSpecAlDWPtr)) {
    goto END;
}

// Get the size of the input color spec array

if( error = GetArray1DSize(pColSpectrumRef, &lColSpectrumDimLW) ) {

    goto END;

}

if ( (lColImSpecAlDWPtr=(double *) malloc(lColSpectrumDimLW * sizeof(double)) ) ==
    NULL ) {

    error=ERR_OUTOFMEMORY;
    goto END;
}

// compute the parameters for the color table

lNbColFeature =lColSpectrumDimLW;

lNbHueBin=(lNbColFeature -2) /2;

lHueStepDW=255.0/(double) lNbHueBin;

// moving filtering the template's color spectrum

switch(lColSpectrumDimLW) {

    case kColorBinNbMedium:
```

```
    for (i=2; i<lNbColFeature-4; i+=2) {  
        lColTempSpreaded[i]= 0.8* lColTempSpecA1DWPtr[i] + 0.1*  
lColTempSpecA1DWPtr[i-2] + 0.1* lColTempSpecA1DWPtr[i+2];  
        lColTempSpreaded[i+1]= 0.8*lColTempSpecA1DWPtr[i+1] + 0.1*  
lColTempSpecA1DWPtr[i-1] + 0.1* lColTempSpecA1DWPtr[i+3];  
    }  
  
    lColTempSpreaded[0]= 0.8* lColTempSpecA1DWPtr[0] + 0.1*  
lColTempSpecA1DWPtr[lNbColFeature-4] + 0.1* lColTempSpecA1DWPtr[2];  
    lColTempSpreaded[1]= 0.8*lColTempSpecA1DWPtr[1] + 0.1*  
lColTempSpecA1DWPtr[lNbColFeature-3] + 0.1* lColTempSpecA1DWPtr[3];  
    lColTempSpreaded[lNbColFeature-4]= 0.8* lColTempSpecA1DWPtr  
[lNbColFeature-4] + 0.1* lColTempSpecA1DWPtr[lNbColFeature-6] + 0.1*  
lColTempSpecA1DWPtr[0];  
    lColTempSpreaded[lNbColFeature-3]= 0.8*lColTempSpecA1DWPtr  
[lNbColFeature-3] + 0.1* lColTempSpecA1DWPtr[lNbColFeature-5] + 0.1*  
lColTempSpecA1DWPtr[1];  
  
    break;  
  
    case kColorBinNbHigh:  
        for (i=4; i<lNbColFeature-6; i+=2) {  
            lColTempSpreaded[i]= 0.7* lColTempSpecA1DWPtr[i] + 0.1*  
lColTempSpecA1DWPtr[i-2] + 0.1* lColTempSpecA1DWPtr[i+2]  
                                + 0.05* lColTempSpecA1DWPtr[i-4] + 0.05*  
lColTempSpecA1DWPtr[i+4];  
            lColTempSpreaded[i+1]= 0.7*lColTempSpecA1DWPtr[i+1] + 0.1*  
lColTempSpecA1DWPtr[i-1] + 0.1* lColTempSpecA1DWPtr[i+3]  
                                + 0.05* lColTempSpecA1DWPtr[i-3] + 0.05*  
lColTempSpecA1DWPtr[i+5];  
        }  
  
        lColTempSpreaded[0]= 0.7* lColTempSpecA1DWPtr[0] + 0.1*  
lColTempSpecA1DWPtr[lNbColFeature-4] + 0.1* lColTempSpecA1DWPtr[2]  
                                + 0.05* lColTempSpecA1DWPtr[lNbColFeature-6]+ 0.  
05* lColTempSpecA1DWPtr[4];  
        lColTempSpreaded[1]= 0.7*lColTempSpecA1DWPtr[1] + 0.1*  
lColTempSpecA1DWPtr[lNbColFeature-3] + 0.1* lColTempSpecA1DWPtr[3]  
                                + 0.05* lColTempSpecA1DWPtr[lNbColFeature-5]+  
0.05* lColTempSpecA1DWPtr[5];  
        lColTempSpreaded[2]= 0.7* lColTempSpecA1DWPtr[2] + 0.1*  
lColTempSpecA1DWPtr[0] + 0.1* lColTempSpecA1DWPtr[4]  
                                + 0.05* lColTempSpecA1DWPtr[lNbColFeature-4]+ 0.  
05* lColTempSpecA1DWPtr[6];  
        lColTempSpreaded[3]= 0.7*lColTempSpecA1DWPtr[3] + 0.1*  
lColTempSpecA1DWPtr[1] + 0.1* lColTempSpecA1DWPtr[5]  
                                + 0.05* lColTempSpecA1DWPtr[lNbColFeature-3]+  
0.05* lColTempSpecA1DWPtr[7];  
        lColTempSpreaded[lNbColFeature-4]= 0.7* lColTempSpecA1DWPtr  
[lNbColFeature-4] + 0.1* lColTempSpecA1DWPtr[lNbColFeature-6] + 0.1*  
lColTempSpecA1DWPtr[0]  
+ 0.05* lColTempSpecA1DWPtr[lNbColFeature-8]+ 0.05* lColTempSpecA1DWPtr[2];
```

```

        lColTempSpreaded[lNbColFeature-3]= 0.7*lColTempSpecAlDWPtr
        [lNbColFeature-3] + 0.1* lColTempSpecAlDWPtr[lNbColFeature-5] + 0.1*
        lColTempSpecAlDWPtr[1]
+ 0.05* lColTempSpecAlDWPtr[lNbColFeature-7]+ 0.05* lColTempSpecAlDWPtr[3];

        lColTempSpreaded[lNbColFeature-6]= 0.7* lColTempSpecAlDWPtr
        [lNbColFeature-6] + 0.1* lColTempSpecAlDWPtr[lNbColFeature-8] + 0.1*
        lColTempSpecAlDWPtr[lNbColFeature-4]
+ 0.05* lColTempSpecAlDWPtr[lNbColFeature-10]+ 0.05* lColTempSpecAlDWPtr[0];

        lColTempSpreaded[lNbColFeature-5]= 0.7*lColTempSpecAlDWPtr
        [lNbColFeature-5] + 0.1* lColTempSpecAlDWPtr[lNbColFeature-7] + 0.1*
        lColTempSpecAlDWPtr[lNbColFeature-3]
+ 0.05* lColTempSpecAlDWPtr[lNbColFeature-9]+ 0.05* lColTempSpecAlDWPtr[1];

        break;

        case kColorBinNbLow:
        default:

        for (i=0; i<lNbColFeature-2; i+=2) {

                lColTempSpreaded[i]= lColTempSpecAlDWPtr[i];

                lColTempSpreaded[i+1]= lColTempSpecAlDWPtr[i+1];

        }

        break;

}

// fuzzy a little bit for saturation too

for (i=0; i<lNbColFeature-2; i+=2) {

        lColTempSpreaded2[i]= ColorSatRemain*lColTempSpreaded[i] + ColorSatSpread *
lColTempSpreaded[i+1];
        lColTempSpreaded2[i+1]= ColorSatRemain*lColTempSpreaded[i+1] + ColorSatSpread
* lColTempSpreaded[i];

}

// no fuzzy for black and white

lColTempSpreaded2[lNbColFeature-2]=lColTempSpecAlDWPtr[lNbColFeature-2];

lColTempSpreaded2[lNbColFeature-1]=lColTempSpecAlDWPtr[lNbColFeature-1];

// if ROI input connected, calculate the color feature based on the ROI image
if( pRoi->numContours>0) {

        // calculate the color match for each subROI

        lROIContourIndex=0;

        jj=lNbROIContour;

```

```
while(jj--) {

    /*Initilization */
    for (i=0; i<lNbColFeature; i++) {
        lColImSpecA1DWPtr[i]=0;
    }

    // create a temporary mask image
    if ( error= NewImage( &lMaskImagePtr, IMAGE_U8, 4, 4, 0) ) {
        goto END;
    }

// ROIContourToMask(const ROI* _roi, int _contourNdx, int _fillValue, GRImage*
_maskImage);
    if (error =ROIContourToMask(pRoi, lROIContourIndex, 1, lMaskImagePtr)) {
        goto END;
    };

    lMskLineWidth = lMaskImagePtr->lineWidth;
    CalcMaskPart(pSrcIMRef, lMaskImagePtr, &lMaskPart);
    GetConstImagePixel(pSrcIMRef, lMaskPart.startImageX, lMaskPart.startImageY, &
lSrcPixPtr);
    GetConstImagePixel(lMaskImagePtr, lMaskPart.startMaskX, lMaskPart.startMaskY,
(ConstPixelPtr*)&lMaskLinePtr);

    lSizeXLW = lMaskPart.sizeX;
    lSizeY LW = lMaskPart.sizeY;

    lImageAreaDW = 0; /* init the Area */

    // control negative sizes
    if (lSizeXLW < 0)
        lSizeXLW = 0;
    if (lSizeY LW < 0)
        lSizeY LW = 0;

    switch (lSrcIMPtr->imageType) {
        case IMAGE_RGB32:
            lSrcPixRGBPtr=lSrcPixPtr.PixRGB_Ptr;
            break;
        case IMAGE_HSL32:
            lSrcPixHSLPtr=lSrcPixPtr.PixHSL_Ptr;
```





```

        if (l1gt < lBlkThreshold ) {
            lNumColorEntry= (lNbColFeature-2);
        }
        else {
            // offset the red color to the center of the first
bin
            lhue = ( lhue + OffsetColor) % 255;
            // clustering non-black-white color feature
            lHueIndex= (double) lhue / lHueStepDW;
            lNumColorEntry= (lsat < pSatThreshold) ? (lHueIndex*
2) :(lHueIndex*2+1);

        }
    }

    lColImSpecA1DWPptr[lNumColorEntry] += 1.0;

    lImageAreaDW++;
} // end of if (*lMaskPixPtr++)

    lBufSrcRGBPptr++;
}

    lSrcPixRGBPptr +=lSrcLineWidth;
    lMaskLinePtr += lMskLineWidth;

}

// calculate the discrete color spectrum
// just in case, lImageAreaDW should never be 0
if(lImageAreaDW==0 )
    for( i=0; i< lNbColFeature; i++ )
        lColImSpecA1DWPptr[i] =0;
else
    for( i=0; i< lNbColFeature; i++ )
        lColImSpecA1DWPptr[i] /= lImageAreaDW;

break;

case IMAGE_HSL32:
    for (lLineLW = 0; lLineLW < lSizeYLW; lLineLW++) {

```

```

lBufSrcHSLPtr=lSrcPixHSLPtr;
lMaskPixPtr = lMaskLinePtr;

for (lColLW = 0; lColLW < lSizeXLW; lColLW++) {

    if (*lMaskPixPtr++) {

        // initializatin

        lhue=lBufSrcHSLPtr->Byte_HSL.H;
        lsat=lBufSrcHSLPtr->Byte_HSL.S;
        llgt=lBufSrcHSLPtr->Byte_HSL.L;

        lNumColorEntry=0;

        if( lsat< BWHueThreshold ) {

            lNumColorEntry=(llgt < BWLgtMiddle ) ? (lNbColFeature-2):
(lNbColFeature-1);

        }

        else {

            // Threshold Black color using Nobelist's formular

            // lBlkThreshold= (long) ((128 - BlkLgtThreshold)*exp(-0.025*
(lsat - BWHueThreshold)) + BlkLgtThreshold );

            if(lsat < 200 )
                lBlkThreshold= (long) ((128 - BlkLgtThreshold) *
expLookup[lsat - BWHueThreshold] + BlkLgtThreshold );
            else
                lBlkThreshold=BlkLgtThreshold;

            if (llgt < lBlkThreshold ) {

                lNumColorEntry= (lNbColFeature-2);

            }

            else {

                // offset the red color to the center of the first bin

                lhue = ( lhue + OffsetColor) % 255;

                // clustering non-black-white color feature

                lHueIndex= (double) lhue / lHueStepDW;

                lNumColorEntry= (lsat < pSatThreshold) ? (lHueIndex*2) :
(lHueIndex*2+1);

            }

        }

        lColImSpecAlDWPPtr[lNumColorEntry] += 1.0;

```

```

        lImageAreaDW++;

    } // end of if (*lMaskPixPtr++)

    lBufSrcHSLPtr++;

}

lSrcPixHSLPtr+=lSrcLineWidth;
lMaskLinePtr += lMskLineWidth;

}

// calculate the discrete color spectrum

// just in case, lImageAreaDW should never be 0
if(lImageAreaDW==0 )
    for( i=0; i< lNbColFeature; i++ )
        lColImSpecA1DWPtr[i] =0;
else
    for( i=0; i< lNbColFeature; i++ )
        lColImSpecA1DWPtr[i] /= lImageAreaDW;

break;

} // end of switch (lSrcIMPtr->imageType)

// moving filtering input image's color spectrum
switch(lColSpectrumDimLW) {
    case kColorBinNbMedium:
        for (i=2; i<lNbColFeature-4; i+=2) {
            lColImSpreaded[i] = 0.8* lColImSpecA1DWPtr[i] + 0.1*
lColImSpecA1DWPtr[i-2] + 0.1* lColImSpecA1DWPtr[i+2]; ✓
            lColImSpreaded[i+1] = 0.8* lColImSpecA1DWPtr[i+1] + 0.1*
lColImSpecA1DWPtr[i-1] + 0.1* lColImSpecA1DWPtr[i+3]; ✓
        }

        lColImSpreaded[0] = 0.8* lColImSpecA1DWPtr[0] + 0.1*
lColImSpecA1DWPtr[lNbColFeature-4] + 0.1* lColImSpecA1DWPtr[2]; ✓
        lColImSpreaded[1] = 0.8* lColImSpecA1DWPtr[1] + 0.1*
lColImSpecA1DWPtr[lNbColFeature-3] + 0.1* lColImSpecA1DWPtr[3]; ✓
        lColImSpreaded[lNbColFeature-4] = 0.8* lColImSpecA1DWPtr
[lNbColFeature-4] + 0.1* lColImSpecA1DWPtr[lNbColFeature-6] + 0.1* lColImSpecA1DWPtr ✓
[0]; ✓
        lColImSpreaded[lNbColFeature-3] = 0.8* lColImSpecA1DWPtr
[lNbColFeature-3] + 0.1* lColImSpecA1DWPtr[lNbColFeature-5] + 0.1* lColImSpecA1DWPtr ✓
[1]; ✓

```

```

        break;

        case kColorBinNbHigh:

            for (i=4; i<lNbColFeature-6; i+=2) {

                lColImSpreaded[i] = 0.7* lColImSpecA1DWPtr[i] + 0.1*
lColImSpecA1DWPtr[i-2] + 0.1* lColImSpecA1DWPtr[i+2]
                + 0.05* lColImSpecA1DWPtr[i-4] + 0.05*
lColImSpecA1DWPtr[i+4];

                lColImSpreaded[i+1] = 0.7* lColImSpecA1DWPtr[i+1] + 0.1*
lColImSpecA1DWPtr[i-1] + 0.1* lColImSpecA1DWPtr[i+3]
                + 0.05* lColImSpecA1DWPtr[i-3] + 0.05*
lColImSpecA1DWPtr[i+5];

            }

            lColImSpreaded[0] = 0.7* lColImSpecA1DWPtr[0] + 0.1*
lColImSpecA1DWPtr[lNbColFeature-4] + 0.1* lColImSpecA1DWPtr[2]
            + 0.05* lColImSpecA1DWPtr[lNbColFeature-6] + 0.05*
* lColImSpecA1DWPtr[4];

            lColImSpreaded[1] = 0.7* lColImSpecA1DWPtr[1] + 0.1*
lColImSpecA1DWPtr[lNbColFeature-3] + 0.1* lColImSpecA1DWPtr[3]
            + 0.05* lColImSpecA1DWPtr[lNbColFeature-5] + 0.05*
* lColImSpecA1DWPtr[5];

            lColImSpreaded[2] = 0.7* lColImSpecA1DWPtr[2] + 0.1*
lColImSpecA1DWPtr[0] + 0.1* lColImSpecA1DWPtr[4]
            + 0.05* lColImSpecA1DWPtr[lNbColFeature-4] + 0.05*
* lColImSpecA1DWPtr[6];

            lColImSpreaded[3] = 0.7* lColImSpecA1DWPtr[3] + 0.1*
lColImSpecA1DWPtr[1] + 0.1* lColImSpecA1DWPtr[5]
            + 0.05* lColImSpecA1DWPtr[lNbColFeature-3] + 0.05*
* lColImSpecA1DWPtr[7];

            lColImSpreaded[lNbColFeature-4] = 0.7* lColImSpecA1DWPtr
[lNbColFeature-4] + 0.1* lColImSpecA1DWPtr[lNbColFeature-6] + 0.1* lColImSpecA1DWPtr
[0]
            + 0.05* lColImSpecA1DWPtr
[lNbColFeature-8] + 0.05* lColImSpecA1DWPtr[2];

            lColImSpreaded[lNbColFeature-3] = 0.7* lColImSpecA1DWPtr
[lNbColFeature-3] + 0.1* lColImSpecA1DWPtr[lNbColFeature-5] + 0.1* lColImSpecA1DWPtr
[1]
+ 0.05* lColImSpecA1DWPtr[lNbColFeature-7] + 0.05* lColImSpecA1DWPtr[3];

            lColImSpreaded[lNbColFeature-6] = 0.7* lColImSpecA1DWPtr
[lNbColFeature-6] + 0.1* lColImSpecA1DWPtr[lNbColFeature-8] + 0.1* lColImSpecA1DWPtr
[lNbColFeature-4]
            + 0.05* lColImSpecA1DWPtr
[lNbColFeature-10] + 0.05* lColImSpecA1DWPtr[0];

            lColImSpreaded[lNbColFeature-5] = 0.7* lColImSpecA1DWPtr
[lNbColFeature-5] + 0.1* lColImSpecA1DWPtr[lNbColFeature-7] + 0.1* lColImSpecA1DWPtr

```

```

    [lNbColFeature-3]
+ 0.05* lColImSpecAlDWPtr[lNbColFeature-9]+ 0.05* lColImSpecAlDWPtr[1];

    break;

    case kColorBinNbLow:
    default:

    for (i=0; i<lNbColFeature-2; i+=2) {

        lColImSpreaded[i] = lColImSpecAlDWPtr[i];
        lColImSpreaded[i+1] = lColImSpecAlDWPtr[i+1];

    }

    break;

}

// fuzzy a little bit for saturation too
for (i=0; i<lNbColFeature-2; i+=2) {

    lColImSpreaded2[i] =ColorSatRemain*lColImSpreaded[i] + ColorSatSpread*
lColImSpreaded[i+1];
    lColImSpreaded2[i+1] =ColorSatRemain*lColImSpreaded[i+1] + ColorSatSpread *
lColImSpreaded[i];

}

// no fuzzy for black and white
lColImSpreaded2[lNbColFeature-2] =lColImSpecAlDWPtr[lNbColFeature-2];
lColImSpreaded2[lNbColFeature-1] = lColImSpecAlDWPtr[lNbColFeature-1];

// calculate the color distance
lColDiff=0;
for (i=0; i<lNbColFeature; i++) {

    lColDiff += fabs(lColTempSpreaded2[i] - lColImSpreaded2[i] );

    // histogram instersection --- if # of bin is large enough

    // lColDiff += (lColTempSpreaded2[i] <= lColImSpreaded2[i]) ?
lColTempSpreaded2[i] : lColImSpreaded2[i] ;

}

if(lImageAreaDW==0 ) // this should never happen !

```

```

        lColDiff=4;

        if ( lColDiff < *ColTolerance *2.0 )
            lMatchFlagPtr[lROIContourIndex]=1;
        else
            lMatchFlagPtr[lROIContourIndex]=0;

        lMatchScorePtr[lROIContourIndex] = (2 - lColDiff) *500; // get the magic
score number

        //destory the temporary mask image
        if (lMaskImagePtr !=NULL )
            DisposeImage(lMaskImagePtr);
        // get next contour index
        lROIContourIndex++;

    } // end of while(lNbROIContour--)

} // end of ROI connected case

else { // ROI not connected, take care of the entire image now
    /*Initilization */
    for (i=0; i<lNbColFeature; i++) {
        lColImSpecAlDWPtr[i]=0;
    }

    // there is only one element in the match flag and match score array
    lROIContourIndex=0;
    GetConstImagePixel(pSrcIMRef,0,0, &lSrcPixPtr);
    lSizeXLW = lSrcIMPtr->xRes;
    lSizeYWL = lSrcIMPtr->yRes;
    lImageAreaDW = (double) lSizeXLW * lSizeYWL;
    switch (lSrcIMPtr->imageType) {
        case IMAGE_RGB32:
            lSrcPixRGBPtr=lSrcPixPtr.PixRGB_Ptr;
            break;
        case IMAGE_HSL32:
            lSrcPixHSLPtr=lSrcPixPtr.PixHSL_Ptr;

```



```

        }
        else {
            // offset the red color to the center of the first
bin
            lhue = ( lhue + OffsetColor) % 255;
            // clustering non-black-white color feature
            lHueIndex= (double) lhue / lHueStepDW;
            lNumColorEntry= (lsat < pSatThreshold) ? (lHueIndex*
2) :(lHueIndex*2+1);

        }
    }

    lColImSpecA1DWPptr[lNumColorEntry] += 1.0;
    lBufSrcRGBPtr++;
}
lSrcPixRGBPtr +=lSrcLineWidth;
}
// calculate the discrete color spectrum
// just in case, lImageAreaDW should never be 0
if(lImageAreaDW==0 )
    for( i=0; i< lNbColFeature; i++ )
        lColImSpecA1DWPptr[i] =0;
else
    for( i=0; i< lNbColFeature; i++ )
        lColImSpecA1DWPptr[i] /= lImageAreaDW;

break;
case IMAGE_HSL32:
    for (lLineLW = 0; lLineLW < lSizeY LW; lLineLW++) {
        lBufSrcHSLPtr=lSrcPixHSLPtr;
        for (lColLW = 0; lColLW < lSizeXLW; lColLW++) {
            // initializatin
            lhue=lBufSrcHSLPtr->Byte_HSL.H;
            lsat=lBufSrcHSLPtr->Byte_HSL.S;
            llgt=lBufSrcHSLPtr->Byte_HSL.L;

            lNumColorEntry=0;

```



```

        if( lsat< BWHueThreshold ) {
            lNumColorEntry=(llgt < BWLgtMiddle ) ? (lNbColFeature-2):
(lNbColFeature-1);
        }

        else {
            // Threshold Black color using Nobelist's formular
            // lBlkThreshold= (long) ((128 - BlkLgtThreshold)*exp(-0.025*
(lsat - BWHueThreshold)) + BlkLgtThreshold );
            if( lsat < 200 )
                lBlkThreshold= (long) ((128 - BlkLgtThreshold) *
expLookup[lsat - BWHueThreshold] + BlkLgtThreshold );
            else
                lBlkThreshold=BlkLgtThreshold;
            if (llgt < lBlkThreshold ) {
                lNumColorEntry= (lNbColFeature-2);
            }
            else {
                // offset the red color to the center of the first bin
                lhue = ( lhue + OffsetColor) % 255;
                // clustering non-black-white color feature
                lHueIndex= (double) lhue / lHueStepDW;
                lNumColorEntry= (lsat < pSatThreshold) ? (lHueIndex*2) :
(lHueIndex*2+1);
            }
        }

        lColImSpecA1DWPptr[lNumColorEntry] += 1.0;

        lBufSrcHSLPtr++;
    }
    lSrcPixHSLPtr+=lSrcLineWidth;
}

// calculate the discrete color spectrum
// just in case, lImageAreaDW should never be 0
if(lImageAreaDW==0 )
    for( i=0; i< lNbColFeature; i++ )
        lColImSpecA1DWPptr[i] =0;
else

```

```

        for( i=0; i< lNbColFeature; i++ )
            lColImSpecA1DWPtr[i] /= lImageAreaDW;

        break;

    } // end of switch (lSrcIMPtr->imageType)

    // moving filtering input image's color spectrum
    switch(lColSpectrumDimLW) {
        case kColorBinNbMedium:
            for (i=2; i<lNbColFeature-4; i+=2) {
                lColImSpreaded[i] = 0.8* lColImSpecA1DWPtr[i] + 0.1*
lColImSpecA1DWPtr[i-2] + 0.1* lColImSpecA1DWPtr[i+2]; ✓
                lColImSpreaded[i+1] = 0.8* lColImSpecA1DWPtr[i+1] + 0.1*
lColImSpecA1DWPtr[i-1] + 0.1* lColImSpecA1DWPtr[i+3]; ✓
            }

            lColImSpreaded[0] = 0.8* lColImSpecA1DWPtr[0] + 0.1*
lColImSpecA1DWPtr[lNbColFeature-4] + 0.1* lColImSpecA1DWPtr[2]; ✓
            lColImSpreaded[1] = 0.8* lColImSpecA1DWPtr[1] + 0.1*
lColImSpecA1DWPtr[lNbColFeature-3] + 0.1* lColImSpecA1DWPtr[3]; ✓
            lColImSpreaded[lNbColFeature-4] = 0.8* lColImSpecA1DWPtr
[lNbColFeature-4] + 0.1* lColImSpecA1DWPtr[lNbColFeature-6] + 0.1* lColImSpecA1DWPtr ✓
[0]; ✓
            lColImSpreaded[lNbColFeature-3] = 0.8* lColImSpecA1DWPtr
[lNbColFeature-3] + 0.1* lColImSpecA1DWPtr[lNbColFeature-5] + 0.1* lColImSpecA1DWPtr ✓
[1]; ✓

            break;

        case kColorBinNbHigh:
            for (i=4; i<lNbColFeature-6; i+=2) {
                lColImSpreaded[i] = 0.7* lColImSpecA1DWPtr[i] + 0.1*
lColImSpecA1DWPtr[i-2] + 0.1* lColImSpecA1DWPtr[i+2] ✓
                + 0.05* lColImSpecA1DWPtr[i-4] + 0.05*
lColImSpecA1DWPtr[i+4]; ✓
                lColImSpreaded[i+1] = 0.7* lColImSpecA1DWPtr[i+1] + 0.1*
lColImSpecA1DWPtr[i-1] + 0.1* lColImSpecA1DWPtr[i+3] ✓
                + 0.05* lColImSpecA1DWPtr[i-3] + 0.05*
lColImSpecA1DWPtr[i+5]; ✓
            }

            lColImSpreaded[0] = 0.7* lColImSpecA1DWPtr[0] + 0.1*
lColImSpecA1DWPtr[lNbColFeature-4] + 0.1* lColImSpecA1DWPtr[2] ✓

```

```

        + 0.05* lColImSpecA1DWPtr[lNbColFeature-6]+ 0.05✓
* lColImSpecA1DWPtr[4];

        lColImSpreaded[1] = 0.7* lColImSpecA1DWPtr[1] + 0.1* ✓
lColImSpecA1DWPtr[lNbColFeature-3] + 0.1* lColImSpecA1DWPtr[3]
        + 0.05* lColImSpecA1DWPtr[lNbColFeature-5]+ 0.05✓
* lColImSpecA1DWPtr[5];

        lColImSpreaded[2] = 0.7* lColImSpecA1DWPtr[2] + 0.1* ✓
lColImSpecA1DWPtr[0] + 0.1* lColImSpecA1DWPtr[4]
        + 0.05* lColImSpecA1DWPtr[lNbColFeature-4]+ 0.05✓
* lColImSpecA1DWPtr[6];

        lColImSpreaded[3] = 0.7* lColImSpecA1DWPtr[3] + 0.1* ✓
lColImSpecA1DWPtr[1] + 0.1* lColImSpecA1DWPtr[5]
        + 0.05* lColImSpecA1DWPtr[lNbColFeature-3]+ 0.05✓
* lColImSpecA1DWPtr[7];

        lColImSpreaded[lNbColFeature-4] = 0.7* lColImSpecA1DWPtr ✓
[lNbColFeature-4] + 0.1* lColImSpecA1DWPtr[lNbColFeature-6] + 0.1* lColImSpecA1DWPtr ✓
[0]
        + 0.05* lColImSpecA1DWPtr ✓
[lNbColFeature-8]+ 0.05* lColImSpecA1DWPtr[2];

        lColImSpreaded[lNbColFeature-3] = 0.7* lColImSpecA1DWPtr ✓
[lNbColFeature-3] + 0.1* lColImSpecA1DWPtr[lNbColFeature-5] + 0.1* lColImSpecA1DWPtr ✓
[1]
+ 0.05* lColImSpecA1DWPtr[lNbColFeature-7]+ 0.05* lColImSpecA1DWPtr[3];

        lColImSpreaded[lNbColFeature-6] = 0.7* lColImSpecA1DWPtr ✓
[lNbColFeature-6] + 0.1* lColImSpecA1DWPtr[lNbColFeature-8] + 0.1* lColImSpecA1DWPtr ✓
[lNbColFeature-4]
        + 0.05* lColImSpecA1DWPtr ✓
[lNbColFeature-10]+ 0.05* lColImSpecA1DWPtr[0];

        lColImSpreaded[lNbColFeature-5] = 0.7* lColImSpecA1DWPtr ✓
[lNbColFeature-5] + 0.1* lColImSpecA1DWPtr[lNbColFeature-7] + 0.1* lColImSpecA1DWPtr ✓
[lNbColFeature-3]
+ 0.05* lColImSpecA1DWPtr[lNbColFeature-9]+ 0.05* lColImSpecA1DWPtr[1];

        break;

        case kColorBinNbLow:
        default:

        for (i=0; i<lNbColFeature-2; i+=2) {

                lColImSpreaded[i] = lColImSpecA1DWPtr[i];

                lColImSpreaded[i+1] = lColImSpecA1DWPtr[i+1];

        }

        break;

}

```

```

    // fuzzy a little bit for saturation too
    for (i=0; i<lNbColFeature-2; i+=2) {

        lColImSpreaded2[i] =ColorSatRemain*lColImSpreaded[i] + ColorSatSpread*
lColImSpreaded[i+1];
        lColImSpreaded2[i+1] =ColorSatRemain*lColImSpreaded[i+1] + ColorSatSpread *
lColImSpreaded[i];
    }

    // no fuzzy for black and white
    lColImSpreaded2[lNbColFeature-2] =lColImSpecA1DWPtr[lNbColFeature-2];
    lColImSpreaded2[lNbColFeature-1] = lColImSpecA1DWPtr[lNbColFeature-1];

    // calculate the color distance
    lColDiff=0;
    for (i=0; i<lNbColFeature; i++) {

        lColDiff += fabs(lColTempSpreaded2[i] - lColImSpreaded2[i] );

        // histogram instersection --- if # of bin is large enough
        // lColDiff += (lColTempSpreaded2[i] <= lColImSpreaded2[i]) ?
lColTempSpreaded2[i] : lColImSpreaded2[i] ;

    }

    if(lImageAreaDW==0 ) // this should never happen !
        lColDiff=4;

    if ( lColDiff < *ColTolerance *2.0 )
        lMatchFlagPtr[lROIContourIndex]=1;
    else
        lMatchFlagPtr[lROIContourIndex]=0;

    lMatchScorePtr[lROIContourIndex] = (2 - lColDiff) *500; // get the magic
score number

} // end of if ( pRoi->numContours>0)

END:
    // just do a cleanup

```

```
    if(lColImSpecA1DWPtr !=NULL)
        free(lColImSpecA1DWPtr);

    return error;
}
```